FAILOVER AND LOAD BALANCING

BACKGROUND

1. Field

The disclosure relates to a method, system, and program for failover and load balancing.

2. Description of the Related Art

An I_T nexus is a pairing of an initiator device and a target device. The devices that request input/output (I/O) operations are referred to as initiators and the devices that perform these operations are referred to as targets. For example, a host computer may be an initiator, and a storage device may be a target. The target may include one or more separate storage devices.

A Host Bus Adapter (HBA) is a hardware device that "connects" the operating system and a Small Computer System Interface (SCSI) bus. The HBA manages the transfer of data between the host computer and the communication path. HBA teaming refers to grouping together several HBAs to form a "team," where each HBA in a team is connected and may route data to a particular target. HBA teams may be built on an Internet Small Computer System Interface (iSCSI) (IETF RFC 3347, published February 2003) portal group concept. iSCSI has been defined as a standard by IETF February 2003. A portal group concept may be described as a collection of Network Portals within an iSCSI Network Entity that collectively support the capability of coordinating a session with connections spanning these portals.

HBA teaming may be used with Small Computer System Interface (SCSI) (American National Standards Institute (ANSI) SCSI Controller Commands-2 (SCC-2) NCITS.318:1998) initiators running Windows® 2000, Windows® XP, or Windows® .NET operating systems. The connection recovery strategy of an I_T nexus may be based on multiple Transmission Control Protocol (TCP) connections (Internet Engineering Task Force (IETF) Request for Comments (RFC) 793, published September 1981). That is, packets from iSCSI initiators running on Windows® operating systems are transmitted and/or received through multiple connections between an initiator and a target. If multiple connections to the same target are established within one HBA, then a miniport driver may handle failover (i.e., when one connection fails, routing packets to another connection) and load balancing (i.e., balancing the load among the HBA connections).

However, there is a need in the art for failover and load balancing across several HBAs, each of which may have one or more connections to the same target.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring now to the drawings in which like reference numbers represent corresponding parts throughout:

FIG. 1A illustrates a computing environment in which certain embodiments are implemented;

FIG. 1B illustrates a computing environment in which certain specific embodiments are implemented;

FIG. 2A illustrates, in a block diagram, a Windows® storage device drivers stack with an optional Class lower filter driver;

FIG. 2B illustrates, in a block diagram, a storage device drivers stack with failover and load balancing capabilities that may be used in a SCSI environment in accordance with certain embodiments;

FIG. 3 illustrates, in a block diagram, an example of configuration of an initiator with failover and load balancing capabilities in accordance with certain embodiments;

FIGs. 4A and 4B illustrate operations for secondary storage device stack hiding in accordance with certain embodiments;

FIGs. 5A, 5B, and 5C illustrate operations for a notification mechanism in accordance with certain embodiments; and

FIGs. 6A and 6B illustrate operations for load balancing in accordance with certain embodiments.

DETAILED DESCRIPTION OF THE EMBODIMENTS

In the following description, reference is made to the accompanying drawings which form a part hereof and which illustrate several embodiments. It is understood that other embodiments may be utilized and structural and operational changes may be made without departing from the scope of embodiments.

FIG. 1A illustrates a computing environment in which embodiments may be implemented. A computer 102 acts as an initiator, while data storage 140 acts as a target to form an I_T nexus. Computer 102 includes one or more central processing units

(CPUs) 104, a volatile memory 106, non-volatile storage 108 (e.g., magnetic disk drives, optical disk drives, a tape drive, etc.), operating system 110 (e.g., Windows® 2000, Windows® XP, or Windows® .NET), and one or more network adapters 128. In certain embodiments, each network adapter is a Host Bus Adapter (HBA). A filter driver 112, a miniport driver 114, and an application program 124 further executes in memory 106.

The computer 102 may comprise a computing device known in the art, such as a mainframe, server, personal computer, workstation, laptop, handheld computer, etc. Any CPU 104 and operating system 110 may be used. Programs and data in memory 106 may be swapped into storage 108 as part of memory management operations.

The data storage 140 includes one or more logical units (i.e., "n" logical units, where "n" may be any positive integer value, which in certain embodiments, is less than 128). Merely for ease of understanding, logical unit 0, logical unit 1, and logical unit "n" are illustrated. Each logical unit may be described as a separate storage device. Additionally, a logical unit number (LUN) is associated with each logical device. In certain embodiments, an HBA team is organized based on the target and LUN (i.e., each HBA that can route data to a particular LUN of a target is grouped into one HBA team), and one HBA may belong to different HBA teams.

Each network adapter 128 includes various components implemented in the hardware of the network adapter 128. Each network adapter 128 is capable of transmitting and receiving packets of data over network 176, which may comprise a Local Area Network (LAN), the Internet, a Wide Area Network (WAN), Storage Area Network (SAN), WiFi (Institute of Electrical and Electronics Engineers (IEEE) 802.11b, published September 16, 1999), Wireless LAN (IEEE 802.11b, published September 16, 1999), etc.

Storage drivers 120 execute in memory 106 and include network adapter 128 specific commands to communicate with each network adapter 128 and interface between the operating system 110 and each network adapter 128. A network adapter 128 and storage driver 120 implement logic to process iSCSI packets, where a SCSI command is wrapped in the iSCSI packet, the iSCSI packet is wrapped in a TCP packet., The transport protocol layer unpacks the payload from the received Transmission Control Protocol (TCP) (Internet Engineering Task Force (IETF) Request for Comments (RFC) 793, published September 1981) packet and transfers the data to the storage driver 120

to return to, for example the application program 124. Further, an application program 124 transmitting data transmits the data to the storage driver 120, which then sends the data to the transport protocol layer to package in a TCP/IP packet before transmitting over the network 176.

A bus controller 134 enables each network adapter 128 to communicate on a computer bus 160, which may comprise any bus interface known in the art, such as a Peripheral Component Interconnect (PCI) bus, PCI express bus, Industry Standard Architecture (ISA), Extended ISA, MicroChannel Architecture (MCA), etc. The network adapter 128 includes a physical communication layer 132 for implementing Media Access Control (MAC) functionality to send and receive network packets to and from remote data storages over a network 176. In certain embodiments, the network adapter 128 may implement the Ethernet protocol (IEEE std. 802.3, published March 8, 2002), Fibre Channel (IETF RFC 3643, published December 2003), or any other network communication protocol known in the art.

The storage 108 may comprise an internal storage device or an attached or network accessible storage. Programs in the storage 108 are loaded into the memory 106 and executed by the CPU 104. An input device 150 is used to provide user input to the CPU 104, and may include a keyboard, mouse, pen-stylus, microphone, touch sensitive display screen, or any other activation or input mechanism known in the art. An output device 152 is capable of rendering information transferred from the CPU 104, or other component, such as a display monitor, printer, storage, etc.

In certain embodiments, in addition to the storage drivers 120, the computer 102 may include other drivers.

The network adapter 128 may include additional hardware logic to perform additional operations to process received packets from the computer 102 or the network 176. Further, the network adapter 128 may implement a transport layer offload engine (TOE) to implement the transport protocol layer in the network adapter as opposed to the computer storage driver 120 to further reduce computer 102 processing burdens. Alternatively, the transport layer may be implemented in the storage driver or other drivers 120 (for example, provided by an operating system).

Various structures and/or buffers (not shown) may reside in memory 106 or may be located in a storage unit separate from the memory 106 in certain embodiments.

FIG. 1B illustrates a computing environment in which certain specific embodiments are implemented. In FIG. 1B, computer 102 includes storage drivers 120 interacting with Host Bus Adapters (HBAs) 128a, 128b, and 128c. Each HBA 128a, 128b, and 128c may be described as a network adapter 128 (FIG. 1A). The computer 102 and HBAs 128a, 128b, and 128c may be described as initiator, while the data storage 140 may be described as a target. The data storage 140 illustrates Logical Unit Numbers (LUNs) that represent logical units.

FIG. 2A illustrates, in a block diagram, a Windows® storage device drivers stack 200 with optional Class lower filter driver. The storage device driver stack 200 includes an upper layer protocol (ULP) 210, which may send packets to and receive packets from a storage class driver 212. The storage class driver 212 may send packets to and receive packets from a class lower filter driver 214. The class lower filter driver 214 is optional in certain embodiments. The class lower filter driver 214 may send packets to and receive packets from a port driver 216. The port driver 216 communicates with a miniport driver 218. The miniport driver 218 is able to communicate with the class lower filter driver 214 via callback interfaces which are provided by embodiments.

Embodiments provide a special filter driver based on the class lower filter driver 214 above the port driver 216 to provide for failover and load balance between cross-HBA connections. The filter driver handles HBA context-switching and path control. The filter driver also provides packet distribution. In particular, the filter driver sniffs SCSI Request Blocks (SRBs) packets included in Input/Output Request Packets (IRPs) between the storage class driver 212 and port driver 216. In certain embodiments, the filter driver is implemented as a lower-level class filter driver. The filter driver is notified as soon as a storage device instance is created, and then, the filter driver attaches itself to the storage driver stack 200 under the class lower filter driver 214.

FIG. 2B illustrates, in a block diagram, a storage device drivers stack 250 with failover and load balancing capabilities that may be used in a SCSI environment in accordance with certain embodiments. The storage device drivers stack 250 may be used with a Windows® 2000, Windows® XP or Windows® .NET operating system. The storage device drivers stack 250 includes an upper layer protocol (ULP) 260, which may send IRPs to and receive IRPs from a SCSI disk class driver 262. The SCSI disk class driver 262 may send SRBs/IRPs to and receive SRB/IRP from a lower level class filter driver

264. The lower level class filter driver 264 may send SRBs/IRPs to and receive SRBs/IRPs from a SCSI port driver 266. The SCSI port driver 266 communicates with a SCSI miniport driver 268. The SCSI miniport driver 268 is able to communicate with the lower level class filter driver 264 via callback interfaces.

FIG. 3 illustrates, in a block diagram, an example of configuration of an initiator with failover and load balancing capabilities in accordance with certain embodiments. In FIG. 3, storage class drivers communicate with a filter driver, which communicates with a port/miniport driver. For example, a storage class driver communicates with a filter driver using storage class driver1 device object 310 and filter driver1 device object 312, the filter driver communicates with a port/miniport driver using filter driver1 device object 312 and port/miniport driver device object 316, 314. A miniport driver communicates with an HBA via port driver exported routines. Each HBA has communication paths to one or more targets. For example, HBA1 318 has communication paths to target1 320 and target2 340. HBA2 338 has communication paths to target1 320 and target2 340. HBA3 358 has communication paths to target2 340 and target3 360. Embodiments provide failover and load balancing across HBAs 318, 338, and 358, some of which have connections to a same target. In certain embodiments, the proposed failover and load balancing approach is based on HBA teaming, built on an iSCSI portal group concept.

Again, in certain embodiments, an HBA team is organized based on the target and LUN (i.e., each HBA that can route data to a particular LUN of a target is grouped into one HBA team), and one HBA may belong to different HBA teams. For example, data may pass through data paths including filter driver1 device object 312, filter driver4 device object 332, and filter driver5 device object 352 to a same LUN (not shown) within target2 340. Each of these data paths passes through a different HBA (HBA1 318, HBA2 338, and HBA3 358, respectively), and so HBA1 318, HBA2 338, and HBA3 358 are in one HBA team. That is, for each HBA in an HBA team, data routed to a LUN within target 2 340 may flow through any HBA in the HBA team (for failover) or may flow through each of the HBAs in the HBA team simultaneously (for load balancing). As another example, an HBA team is also formed by HBA1 318 and HBA2 338, because corresponding data paths pass through HBA1 318 and HBA2 338 to a LUN (not shown) in target1 320.

Embodiments provide new callback interfaces for implementing failover and load balancing capabilities in a filter driver.

The failover techniques provide high availability of communication paths to a target when a first HBA or data path thought the first HBA fails (i.e., packets are routed to a second HBA that is connected to the same target). For failover, embodiments provide secondary storage device stack hiding and a notification mechanism.

FIGs. 4A and 4B illustrate operations for secondary storage device stack hiding in accordance with certain embodiments. Control begins at block 400 with a storage device stack being built for a logical unit. In block 402, it is determined whether this is the first storage device stack for this logical unit. If so, processing continues to block 404, otherwise, processing continues to block 404. That is, when there are several HBAs connected to the same target, a storage device stack is created for each SCSI LUN of the target for each of the HBAs to enable the filter driver to handle and redirect SRBs for each HBA. If the file system were to mount on each of the storage device stacks being built for the same target, this can lead to synchronization problems due to multiple accesses to the same storage device instance. Therefore, the first storage device stack built for each LUN becomes a "primary" storage device stack. After that, all other storage device stacks built for the LUN are treated as "secondary" storage device stacks. So, in block 404, the storage device stack is designated as a primary storage device stack. In block 406, the storage device stack is designated as a secondary storage device stack.

For example, the storage device stack of storage class driver1 device object 310, filter driver1 device object 312, port/miniport driver device object 314, 316 is designated a primary storage device stack in this example. The storage device stack of storage class driver 1a device object 330, filter driver4 device object 332, port/miniport driver device object 334, 336 is designated a secondary storage device stack in this example. The storage device stack of storage class driver 1a device object 350, filter driver5 device object 352, port/miniport driver device object 354, 356 is designated a secondary storage device stack in this example. The filter driver enables file system mounting on one, primary storage device stack (e.g., storage device stack 310, 312, 314, and 316) and prevents file system mounting on the other, secondary storage device stacks (e.g., storage device stack 330, 332, 334, and 336 and storage device stack 350, 352, 354, and

356). Likewise, the storage device stack starting with storage class driver2 device object is a primary storage device stack, while the storage device stack starting with storage class driver 2a device object is a secondary storage device stack. Because of secondary storage device stack hiding, data is routed through the primary storage device stacks and not through the secondary storage device stacks.

In FIG. 4B, control begins at block 420 with a packet being completed from a secondary storage device stack with success status. In block 422, the filter driver changes the success status to an error status. In block 424, the filter driver sets the sense key value to not ready. In block 426, the filter driver sets the sense code to indicate that no media is in the storage device. Thus, in certain embodiments, to prevent the file system mounting on the secondary storage device stacks, the filter driver flips the status of SRBs completed from the secondary storage device stacks with success status to error status (e.g., SRB_STATUS_ERROR), sets a sense key value to SCSI_SENSE_NOT_READY and sets an additional sense code to SCSI_ADSENSE_NO_MEDIA_IN_DEVICE.

FIGs. 5A, 5B, and 5C illustrate operations for a notification mechanism in accordance with certain embodiments. The filter driver and miniport driver implement a protocol for interaction with each other.

FIG. 5A illustrates operations for failover processing implemented in a miniport driver in accordance with certain embodiments. In FIG. 5A, control begins at block 500 with receipt of a packet for a particular HBA data path at the miniport driver. The term "HBA data path" may be described as a data path from a miniport via an HBA to a target. One type of packet may be an SRB. In block 502, if the HBA data path has failed, the processing continues to block 504, otherwise, processing continues to block 508.

In block 504, the miniport driver uses the notification callback method to notify the filter driver that the HBA data path has failed. Optionally, the miniport driver provides a new HBA data path identifier to the filter driver, and the filter driver redirects packets to this new HBA data path. For example, for the HBA team formed by HBA1 318, HBA2 338, and HBA3 358, if HBA1 318 fails, the miniport may specify a new HBA data path including either HBA2 338 or HBA3 358. In certain embodiments, a pointer to the notification callback method and a pointer to the filter device extension for a current HBA data path may be sent to the miniport driver via an I/O control (IOCTL) in an add

device method of the filter driver. In certain embodiments, when a HBA data path fails, the miniport driver notifies the filter driver of the HBA data path fail by calling the notification callback method with a status parameter and with a pointer parameter, where the pointer points to the device extension of the filter driver, corresponding to the current path.

The miniport driver does not complete packets with error status as this may cause the port driver to freeze an SRB queue or initiate a bus reset. Instead, in block 506, the miniport driver completes the pending (i.e., outstanding) and newly received packets for the failed HBA data path with a success status. In block 508, since the HBA data path has not failed, the miniport driver sends the packet to the target via the HBA data path.

For example, if a SRB is received at miniport 316 for routing to HBA1 318, and if HBA1 318 has failed, then the miniport driver 316 completes all pending SRBs for HBA1 318 as part of the protocol to notify the filter driver 312 to redirect the SRBs to another HBA. For instance, the filter driver 312 may redirect the SRBs to HBA2 338 or HBA3 358.

FIG. 5B illustrates operations implemented in a filter driver for failover processing in accordance with certain embodiments. Control begins at block 510 with the filter driver receiving a HBA data path failure notification. In block 512, if a new HBA data path identifier is specified by the miniport driver, processing continues to block 514, otherwise, processing continues to block 518. In block 514, the filter driver changes the status of each packet being completed from the failed path from success status to busy status. This change in status causes the class driver to reissue the packets. When the packets are reissued, the filter driver treats them as new requests and, if possible, redirects them to a new path.

In block 516, the filter driver sends packets missed by the class driver to the specified HBA data path. In block 518, since a new path identifier is not specified, then the filter driver selects a new path. In certain embodiments, if there is just one path in an HBA team, the filter driver continues to send new packets to the failed HBA data path (e.g., working in PATH_THROUGH mode).

FIG. 5C illustrates operations implemented in a miniport driver when an HBA is restored in accordance with certain embodiments. Control begins at block 520 with a HBA data path being restored (e.g., an HBA is brought online or an HBA or data path

through this HBA that had previously failed is restored). When a HBA data path has restored, the miniport driver notifies the filter driver by calling the notification callback method (block 522). In certain embodiments, the miniport driver notifies the filter driver of the HBA data path restoration by calling the notification callback method with a status parameter and with a pointer parameter, where the pointer points to the device extension of the filter driver, corresponding to the current path. The miniport driver may also pass the filter driver a new path identifier to which the filter driver should redirect new packets. In block 524, if a new HBA data path is specified, processing continues to block 526, otherwise, processing continues to block 528.

After the restore notification, if a new HBA data path is specified, the filter driver sends the new packets to the new HBA data path (block 526). If no new HBA data path is specified, the filter driver continues to send the packets on the current active path.

In additional to failover processing, embodiments provide a filter driver for balancing of I/O workload across multiple HBA data paths spanning multiple HBAs. Embodiments provide static load balancing and dynamic load balancing.

FIGs. 6A and 6B illustrate operations for load balancing in accordance with certain embodiments. For load balancing, each HBA data path in an HBA team has an associated value, referred to as a load balancing share that represents the percentage of a total I/O workload that the given HBA data path is able to handle.

FIG. 6A illustrates operations for static load balancing in accordance with certain embodiments. Control begins at block 600 with receipt of a data packet to be transmitted at the filter driver. In block 602, the filter driver determines the load balancing share associated with each HBA data path in the HBA team. For static load balancing, the load balancing shares may be specified manually and stored, for example, in a Windows® registry. The miniport driver retrieves the load balancing share values and forwards these to the filter driver via the notification callback method. On receiving the retrieved load balancing share values, the filter driver updates the load balancing shares with new values.

In block 604, the filter driver determines the moving mean data length (MDL) of the transmitted packets for each HBA data path (e.g., SCSI transfer length of the packets). In certain embodiments, load balancing may be based on actual data length of transmitted packets, rather than on a number of packets. In block 606, the filter driver

determines a data quota for each HBA data path. In certain embodiments, the data quota for a HBA data path is the MDL for the HBA data path multiplied by a ratio of the load balancing share for the HBA data path and a minimal value of a load balancing share in the HBA team (i.e., data quota = MDL * (load balancing share/minimal load balancing share in team)).

In certain embodiments, MDL is recalculated for each packet or group of packets, while data quotas are recalculated periodically (e.g., after a certain number of packets are transferred). The periodic intervals may be determined by a product of the number of HBA data paths in the HBA team and a tunable load balancing frequency update factor. The load balancing frequency update factor allows tuning of load balancing and increases performance. The load balancing frequency update factor may be set, for example, by a system administrator. The higher the frequency of updating of data quotas, the less difference between specified and actual load balancing shares. Also, more frequent updates may take up more processor time.

In block 608, the filter driver determines a maximum number of commands for a target logical unit. In block 610, the filter driver selects a HBA data path on which to send the packet using a round robin technique in which packets are sent along one HBA data path until a data quota is reached or a maximum of commands per the target logical unit is reached.

That is, for static load balancing, a round-robin technique may be used. In certain embodiments, valid HBA data paths for a current HBA team are collected in a double-linked list for the round-robin operation. Then, HBA data paths are switched during the load balancing operation using this list. Otherwise, the packet flow switches to the next HBA team member. That is, in certain embodiments, sending of packets for a given path continues until the amount of data transferred reaches the previously calculated data quota for this HBA data path or a maximum number of SCSI commands per a target LUN is reached.

For static load balancing, the actual distribution of packets among the HBA data paths may differ from a specified distribution (i.e., load balancing shares specified by, for example, a system administrator). The less difference between the actual and specified distributions, the higher the quality of the static load balancing. The specified load balancing share values affect the performance of static load balancing.

FIG. 6B illustrates operations for dynamic load balancing in accordance with certain embodiments. Dynamic load balancing avoids congestion on a single HBA data path as long as there is available bandwidth on other paths by dynamically adjusting I/O workload among paths. Thus, dynamic load balancing attempts to enable more efficient use of a storage/network available bandwidth.

Control begins at block 620 with the filter driver receiving a data packet. In block 622, the filter driver determines a data transfer speed for each HBA data path in the HBA team. In certain embodiments, data transfer speed is calculated for each HBA data path as a ratio of total data transferred and total time spent on transferring data. An updating routine determines how frequently the transfer speeds are to be updated. A tunable parameter for dynamic load balancing is a transfer speed update frequency factor, which defines how often the updating routine is to be invoked. The transfer speed update frequency factor may be tuned based on the specific behavior of the delivery subsystem.

In block 624, the filter driver updates the load balancing share for each HBA data path. Load balancing shares for each path in the HBA team may be updated proportionally to their data transfer speed (i.e., data transfer speed for the selected HBA data path divided by HBA team's data transfer speed). In block 626, the filter driver selects a HBA data path on which to send the packet based on the load balancing shares.

While working in failover and load balancing mode, HBA failures disable one or more HBA team members. With embodiments, the remaining HBA team members continue functioning, maintaining the same ratio between the load balancing shares specified.

Thus, embodiments provide a combination of high availability and static and dynamic load balancing. A software module that implements the solution is relatively small and poses minimal overhead on the system. Embodiments provide the ability to quickly and easily turn on and off the functionality provided by embodiments of the solution by insertion/removal of the filter driver to/from the device stack. Embodiments are compatible across various platforms (e.g., Windows® platforms, either 32 bit or 64 bit). Also, embodiments are applicable to any SCSI and/or iSCSI based SAN and/or Network Attached Storage (NAS) system. Moreover, embodiments for failover and load balancing support multiple (two or more) HBAs.

Additional Embodiments Details

The described techniques for failover and load balancing may be implemented as a method, apparatus or article of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof. The term "article of manufacture" as used herein refers to code or logic implemented in hardware logic (e.g., an integrated circuit chip, Programmable Gate Array (PGA), Application Specific Integrated Circuit (ASIC), etc.) or a computer readable medium, such as magnetic storage medium (e.g., hard disk drives, floppy disks,, tape, etc.), optical storage (CD-ROMs, optical disks, etc.), volatile and non-volatile memory devices (e.g., EEPROMs, ROMs, PROMs, RAMs, DRAMs, SRAMs, firmware, programmable logic, etc.). Code in the computer readable medium is accessed and executed by a processor. The code in which preferred embodiments are implemented may further be accessible through a transmission media or from a file server over a network. In such cases, the article of manufacture in which the code is implemented may comprise a transmission media, such as a network transmission line, wireless transmission media, signals propagating through space, radio waves, infrared signals, etc. Thus, the "article of manufacture" may comprise the medium in which the code is embodied. Additionally, the article of manufacture may comprise a combination of hardware and software components in which the code is embodied, processed, and executed. Of course, those skilled in the art recognize that many modifications may be made to this configuration without departing from the scope of embodiments, and that the article of manufacture may comprise any information bearing medium known in the art.

In the described embodiments, certain logic was implemented in a driver. In alternative embodiments, the logic implemented in the driver and/or network adapter may be implemented all or in part in network hardware.

In certain embodiments, the network adapter may be implemented as a PCI card. In alternative embodiments, the network adapter may comprise integrated circuit components mounted on the computer 102 motherboard.

In certain embodiments, the network adapter may be configured to transmit data across a cable connected to a port on the network adapter. In alternative embodiments, the

network adapter embodiments may be configured to transmit data over a wireless network or connection, such as wireless LAN.

The illustrated logic of FIGs. 4A, 4B, 5A, 5B, 5C, 6A, and 6C show certain events occurring in a certain order. In alternative embodiments, certain operations may be performed in a different order, modified or removed. Moreover, operations may be added to the above described logic and still conform to the described embodiments. Further, operations described herein may occur sequentially or certain operations may be processed in parallel. Yet further, operations may be performed by a single processing unit or by distributed processing units.

The foregoing description of various embodiments has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the embodiments to the precise forms disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the embodiments be limited not by this detailed description, but rather by the claims appended hereto. The above specification, examples and data provide a complete description of the manufacture and use of the composition of the embodiments. Since many embodiments can be made without departing from the spirit and scope of the embodiments, the embodiments reside in the claims hereinafter appended.